МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

**«Исследование состояния динамической памяти.»**

Отчет по лабораторной работе № 2 по дисциплине

«Технологии программирования»

Выполнил студент группы: ПИб-2301\_\_\_\_\_\_\_\_\_\_\_\_\_ /Морозов Д.А.

Проверил преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Долженкова М.Л.

**Киров 2023**

**Цели работы:**

* Разработать приложение, реализующее заданную динамическую структуру;
* Изучить дамп памяти при вставке и удалении элементов.

**Задание:**

Разработать консольное приложение, в котором пользователь работает с динамической структурой дек, каждый элемент которого – массив типа char и значение типа int. Работа с памятью должна быть реализована при помощи функций malloc и free.

**Словесное описание алгоритма:**

Алгоритм содержит несколько функций для вставки и удаления узлов в списке:

1. Структура **Node** представляет узел списка и содержит целочисленное значение (**value**), указатель на строку (**str**), указатель на предыдущий узел (**prev**) и указатель на следующий узел (**next**).
2. Функция **createNode** создает новый узел с заданными значениями. Она выделяет память под структуру **Node** с помощью функции **malloc**, копирует строку **str** в отдельно выделенную память и устанавливает указатели **prev** и **next** в **NULL**. Затем функция возвращает указатель на созданный узел.
3. Функция **insertAfter** вставляет новый узел (**newNode**) после указанного узла (**prevNode**). Если **prevNode** равен **NULL**, функция просто возвращает управление без выполнения каких-либо действий. В противном случае, она устанавливает указатель **next** нового узла на следующий узел **prevNode**, указатель **prev** на **prevNode**, а также обновляет соответствующие указатели узла **prevNode** и следующего узла, чтобы они указывали на новый узел.
4. Функция **insertBefore** вставляет новый узел (**newNode**) перед указанным узлом (**nextNode**). Если **nextNode** равен **NULL**, функция вставляет новый узел в начало списка путем обновления указателя **headRef**. В противном случае, она устанавливает указатель **prev** нового узла на предыдущий узел **nextNode**, указатель **next** на **nextNode**, а также обновляет соответствующие указатели узла **nextNode** и предыдущего узла, чтобы они указывали на новый узел.
5. Функция **insertAtHead** вставляет новый узел (**newNode**) в начало списка путем обновления указателя **headRef** и соответствующих указателей у нового узла и предыдущего первого узла.
6. Функция **insertAtTail** вставляет новый узел (**newNode**) в конец списка. Если список пуст (**\*headRef == NULL**), функция устанавливает указатель **headRef** на новый узел. В противном случае, она перемещается до последнего узла в списке и устанавливает указатель **next** последнего узла на новый узел, а указатель **prev** нового узла на последний узел.
7. Функция **deleteNode** удаляет указанный узел (**delNode**) из списка. Если список пуст (**\*headRef == NULL**) или **delNode** равен **NULL**, функция просто возвращает управление без выполнения каких-либо действий. В противном случае, функция проверяет, является ли **delNode** первым узлом списка и обновляет указатель **headRef**, если это так. Затем она обновляет указатели у предыдущего и следующего узлов, чтобы они указывали друг на друга, обходя удаляемый узел, и освобождает память, выделенную под строку и узел.
8. Функция **printList** выводит содержимое списка на экран. Она перемещается по списку, начиная с головного узла (**head**), и выводит номер узла, его значение и строку.
9. В функции **main** реализовано интерактивное взаимодействие с пользователем. Она выводит меню с вариантами операций, принимает выбранный код операции от пользователя и выполняет соответствующую функцию для вставки, удаления или печати списка. При вводе некорректных данных пользователю выводятся сообщения об ошибке.
10. В конце **main** осуществляется освобождение памяти, выделенной для списка, путем прохода по всем узлам и освобождения памяти для каждого узла и его строки.

Данный алгоритм предоставляет базовую реализацию двусвязного списка с возможностью вставки и удаления узлов на разных позициях.

При помощи структуры case пользователь может обращаться к различным функциям посредством определения номера обращения.

**Dump памяти:**

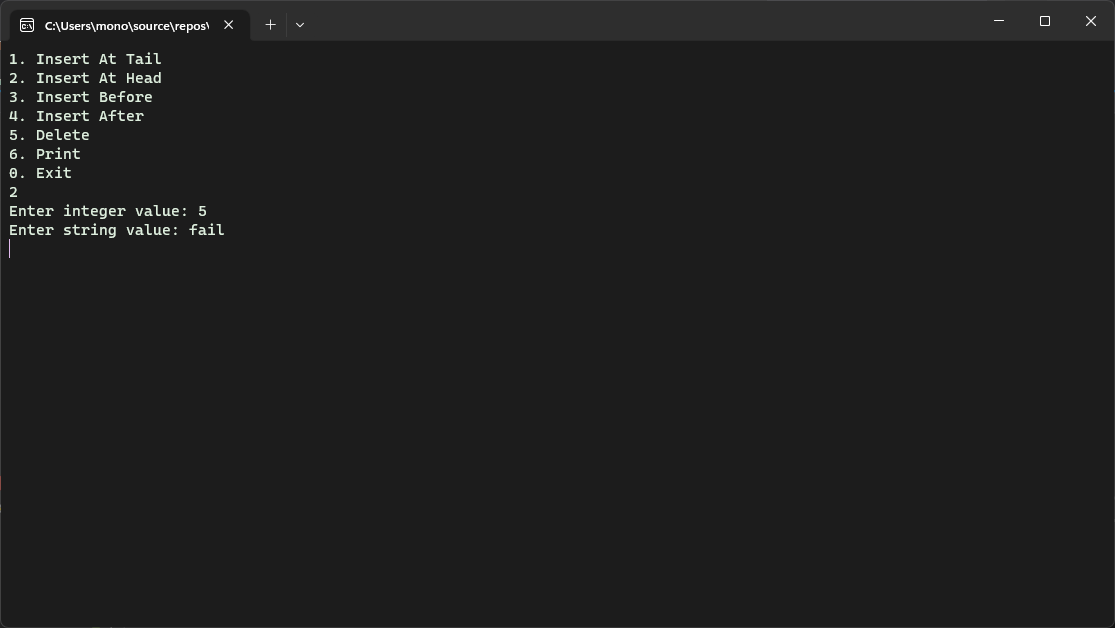


Рис. 1 – введенные данные

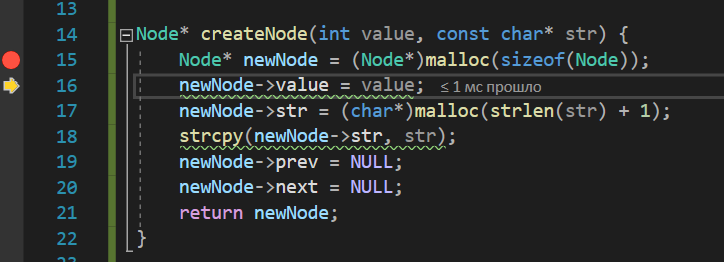


Рис. 2 – выбранная строка кода в отладчике

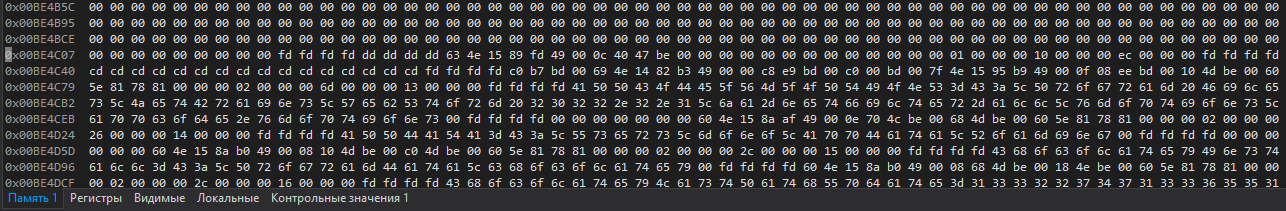


Рис. 3 – dump памяти на момент начала инициализации узла

В выбранной области находится информация об инициализированном узле. Можно заметить, что ячейка памяти, на которой строит курсор выделена с двух сторон некими маркерами (fd fd fd fd). Внутри этих маркеров будут записаны значения, которые хранит в себе узел.

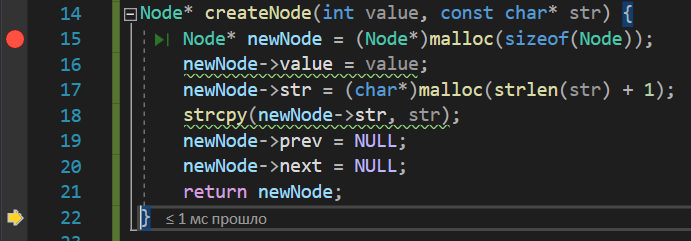


Рис. 4 – выбранная строка кода в отладчике

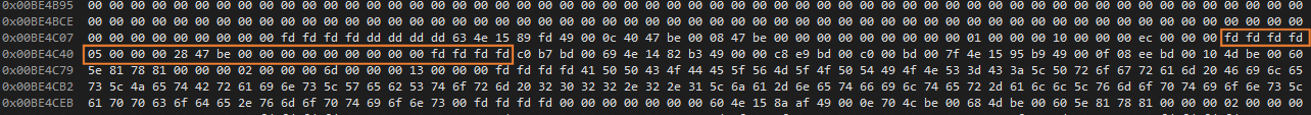


Рис. 5 – dump памяти на момент конца инициализации узла

Можно заметить, что содержимое ячейки памяти изменилось – добавились указатель на нулевой элемент массива, символ и число типа int

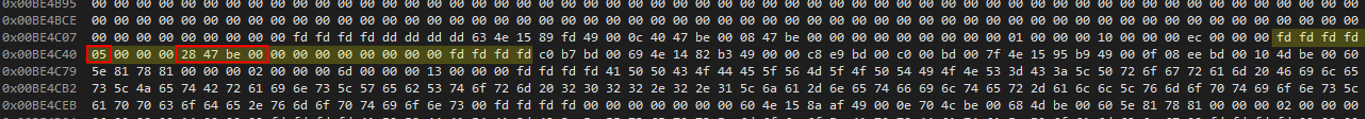


Рис. 6 – число типа int и указатель на нулевой элемент массива из char

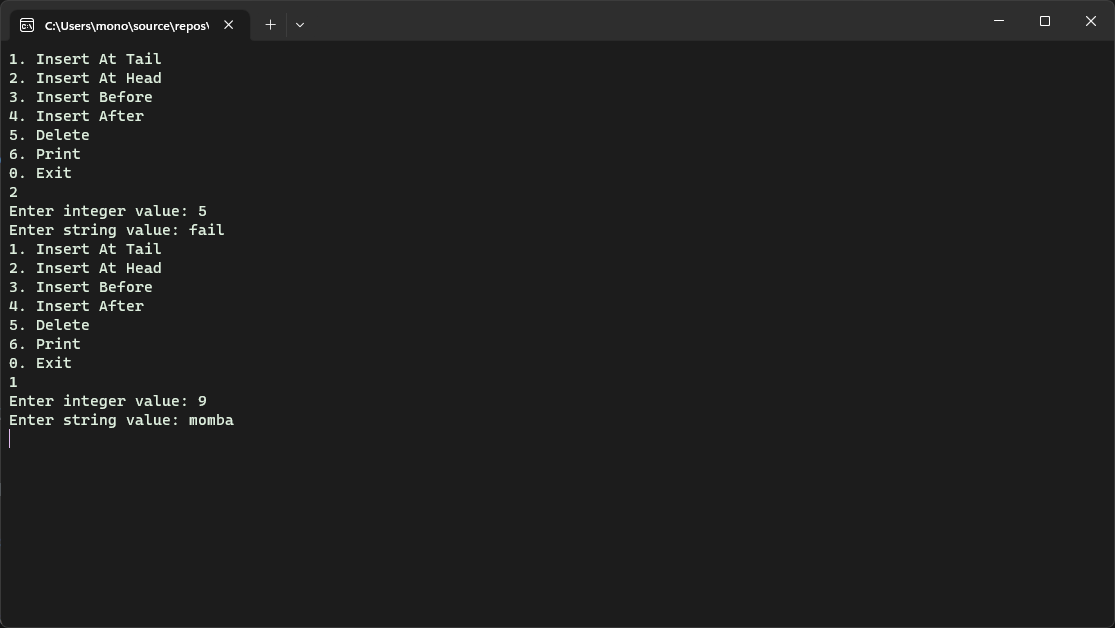


Рис. 7 – добавление нового узла

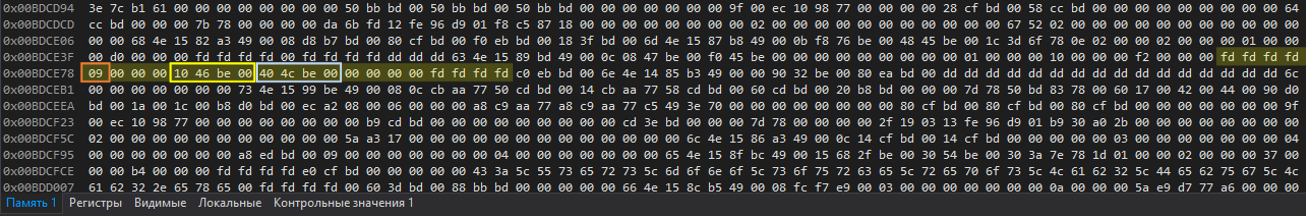


Рис. 8 – dump памяти на момент конца инициализации второго узла (оранжевый – значение int, желтый – ссылка на массив char, голубой – ссылка на следующий элемент в деке)

Можно заметить, что выделенная область перед содержимым объекта изменилась. Эта область, хранящая в себе размер всех узлов двусвязного списка, увеличилась.

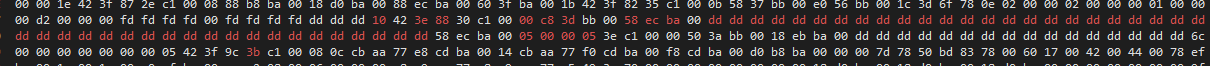


Рис. 9 – dump памяти после удаления элемента

После удаления элемента видно, что память, выделенная для него, не освобождается полностью, но заполняется некоторыми значениями. Это указывает на то, что данная область памяти остается зарезервированной и может быть использована для хранения других данных в будущем. Важно отметить, что эта область памяти может быть заполнена только значениями, переданными программой, и не может быть использована для хранения информации из других приложений или программ. Такая реализация обеспечивает наличие выделенной памяти, которая может быть использована для хранения значений, специфических для данной программы, и не доступна для других приложений или внешних источников данных.

**Листинг кода:**

#include <iostream>

#include <cstdlib>

#include <cstring>

using namespace std;

struct Node {

int value;

char\* str;

Node\* prev;

Node\* next;

};

Node\* createNode(int value, const char\* str) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->value = value;

newNode->str = (char\*)malloc(strlen(str) + 1);

strcpy(newNode->str, str);

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

void insertAfter(Node\*\* headRef, Node\* prevNode, Node\* newNode) {

if (prevNode == NULL) {

return;

}

newNode->next = prevNode->next;

newNode->prev = prevNode;

prevNode->next = newNode;

if (newNode->next != NULL) {

newNode->next->prev = newNode;

}

}

void insertBefore(Node\*\* headRef, Node\* nextNode, Node\* newNode) {

if (nextNode == NULL) {

return;

}

newNode->prev = nextNode->prev;

newNode->next = nextNode;

nextNode->prev = newNode;

if (newNode->prev != NULL) {

newNode->prev->next = newNode;

}

else {

\*headRef = newNode;

}

}

void insertAtHead(Node\*\* headRef, Node\* newNode) {

newNode->next = \*headRef;

if (\*headRef != NULL) {

(\*headRef)->prev = newNode;

}

\*headRef = newNode;

}

void insertAtTail(Node\*\* headRef, Node\* newNode) {

if (\*headRef == NULL) {

\*headRef = newNode;

}

else {

Node\* current = \*headRef;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

void deleteNode(Node\*\* headRef, Node\* delNode) {

if (\*headRef == NULL || delNode == NULL) {

return;

}

if (\*headRef == delNode) {

\*headRef = delNode->next;

}

if (delNode->next != NULL) {

delNode->next->prev = delNode->prev;

}

if (delNode->prev != NULL) {

delNode->prev->next = delNode->next;

}

free(delNode->str);

free(delNode);

}

void printList(Node\* head) {

Node\* current = head;

int count = 0;

while (current != NULL) {

cout << " " << endl;

cout << count++ << " " << current->value << " " << current->str << endl;

current = current->next;

}

cout << " " << endl;

}

int main() {

Node\* head = NULL;

while (true) {

cout << "1. Insert At Tail\n2. Insert At Head\n3. Insert Before\n4. Insert After\n5. Delete\n6. Print\n0. Exit\n";

int code;

while (!(cin >> code)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

if (code == 1) {

cout << "Enter integer value: ";

int value;

while (!(cin >> value)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

cout << "Enter string value: ";

char str[256]; cin >> str;

Node\* node = createNode(value, str);

insertAtTail(&head, node);

}

else if (code == 2) {

cout << "Enter integer value: ";

int value;

while (!(cin >> value)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

cout << "Enter string value: ";

char str[256]; cin >> str;

Node\* node = createNode(value, str);

insertAtHead(&head, node);

}

else if (code == 3) {

if (head == NULL) {

cout << "List is empty.\n";

continue;

}

cout << "Enter location number of the node to insert before: ";

int value;

while (!(cin >> value)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

Node\* current = head;

int i = 0;

while (current != NULL) {

if (i == value) {

cout << "Enter integer value: ";

int newValue;

while (!(cin >> newValue)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

cout << "Enter string value: ";

char newStr[256]; cin >> newStr;

Node\* newNode = createNode(newValue, newStr);

insertBefore(&head, current, newNode);

break;

}

current = current->next;

i++;

}

if (current == NULL) {

cout << "Invalid location number. Please try again.\n";

}

}

else if (code == 4) {

if (head == NULL) {

cout << "List is empty.\n";

continue;

}

cout << "Enter location number of the node to insert after: ";

int locNumber;

while (!(cin >> locNumber)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

int count = 0;

Node\* current = head;

while (current != NULL) {

if (count == locNumber) {

cout << "Enter integer value: ";

int newValue;

while (!(cin >> newValue)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

cout << "Enter string value: ";

char newStr[256]; cin >> newStr;

Node\* newNode = createNode(newValue, newStr);

insertAfter(&head, current, newNode);

break;

}

current = current->next;

count++;

}

if (current == NULL) {

cout << "Node not found at location number " << locNumber << ".\n";

}

}

else if (code == 5) {

if (head == NULL) {

cout << "List is empty.\n";

continue;

}

cout << "Enter location number of the node to delete: ";

int location;

while (!(cin >> location)) {

cout << "Invalid input. Please enter a number.\n";

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

}

if (location < 0) {

cout << "Invalid location.\n";

continue;

}

if (location == 0) {

deleteNode(&head, head);

continue;

}

int count = 1;

Node\* current = head;

Node\* prev = NULL;

while (current != NULL && count != location) {

prev = current;

current = current->next;

count++;

}

if (current == NULL) {

cout << "Location exceeds the length of the list.\n";

continue;

}

deleteNode(&head, current);

}

else if (code == 6) {

if (head == NULL) {

cout << "List is empty.\n";

continue;

}

printList(head);

}

else if (code == 0) {

break;

}

else

{

cout << "Invalid input. Try again.\n";

}

}

// free memory

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

free(current->str);

free(current);

current = next;

}

return 0;

}

**Экранные формы:**

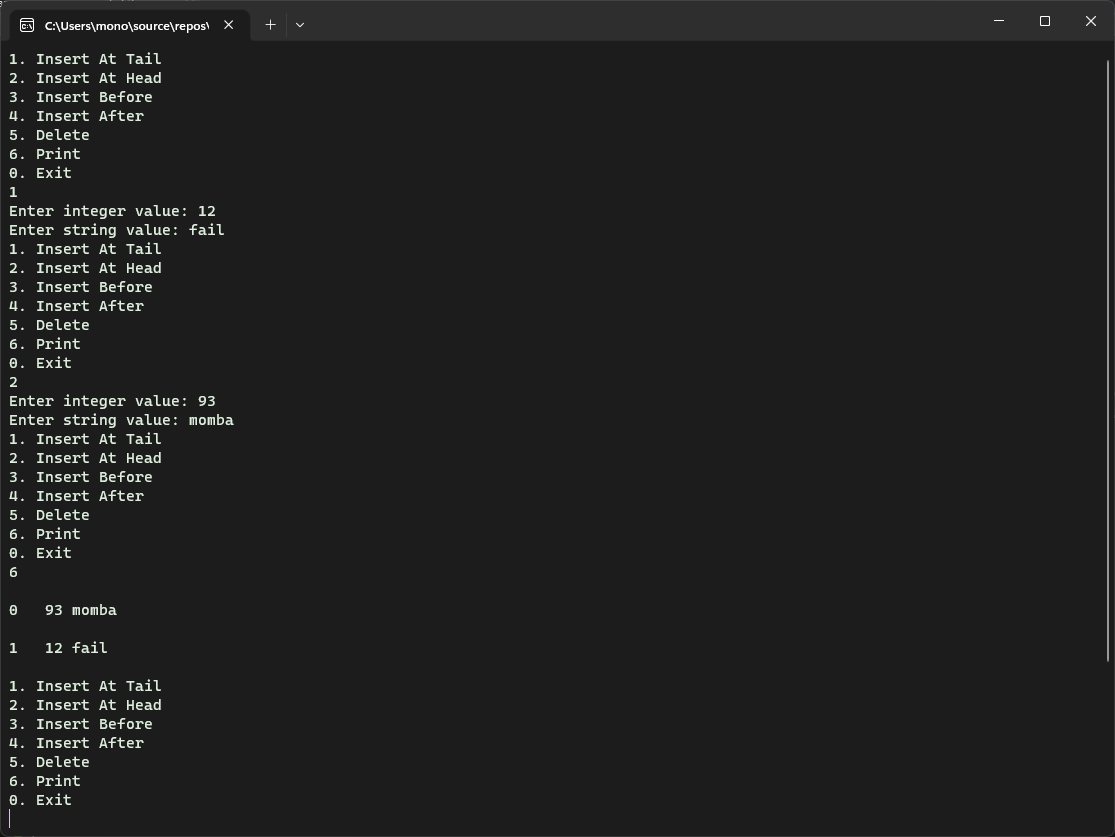
****

Рис. 10 – добавление элементов в дек

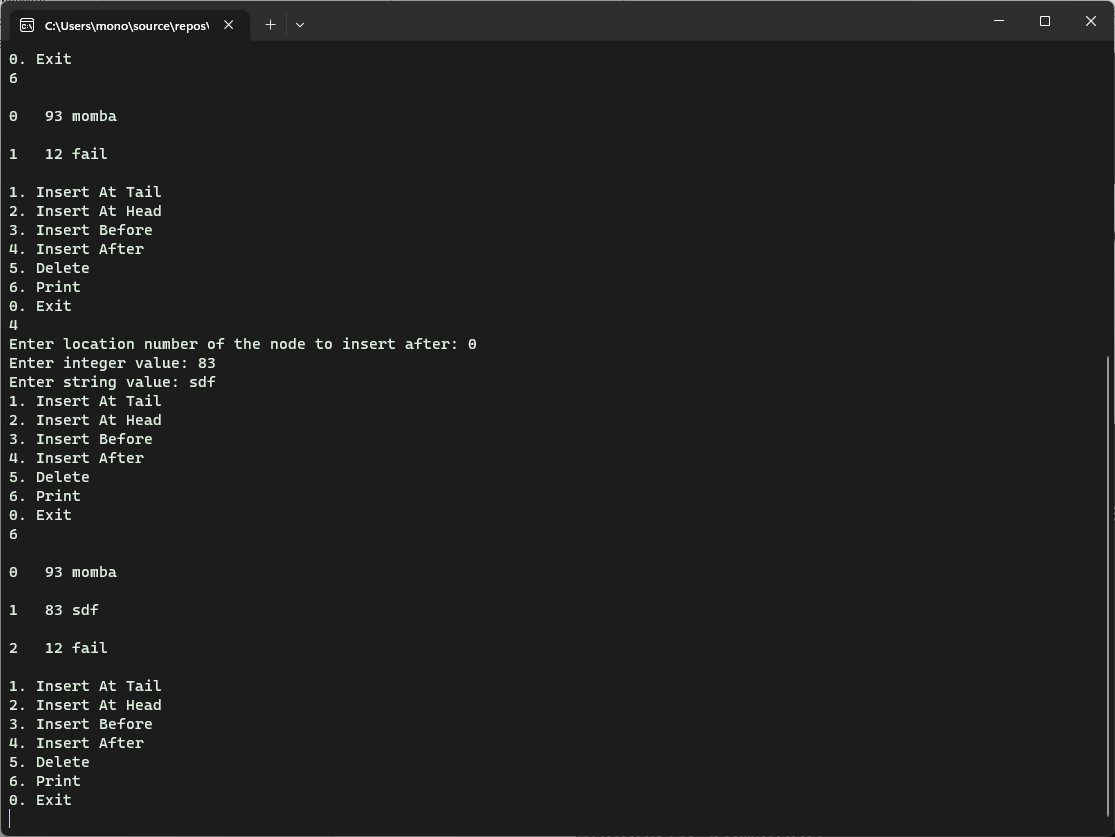
****

Рис. 11 – добавление элемента после указанного индекса

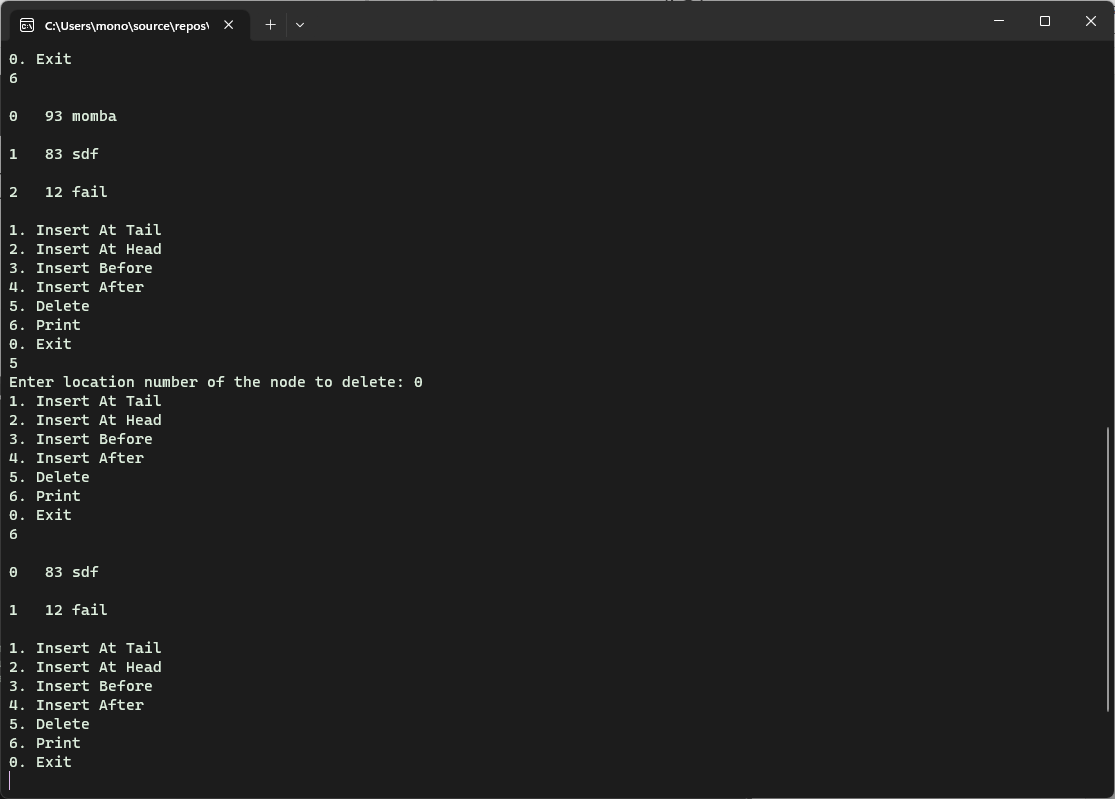


Рис. 12 – удаление по индексу

**Вывод:**

В ходе данной лабораторной работы было разработано консольное приложение, которое реализует динамическую структуру данных - дек (двусторонняя очередь). Каждый элемент дека представляет собой массив символов (тип char) и целочисленное значение (тип int). Реализация дека была осуществлена при помощи struct

Структура используется для определения пользовательского типа данных, который может содержать различные поля разных типов данных. В данном случае, каждый элемент дека представлен структурой Node, которая содержит поле value типа int для хранения целочисленного значения и поле str типа char\* для хранения указателя на динамический массив символов.

Одним из основных аспектов работы с динамическими структурами данных является управление памятью. Просмотр памяти (memory dump) при вставке и удалении элементов позволяет отслеживать использование памяти в программе и выявлять возможные утечки памяти. Путем анализа содержимого памяти можно проверить, правильно ли освобождается память после удаления элементов и корректно ли выделяется память при добавлении новых элементов.

Для управления памятью при добавлении и удалении элементов двусвязного списка malloc и free для выделения и освобождения памяти для динамического массива символов.

Использование структур и динамической памяти позволяет программе гибко управлять данными и эффективно использовать ресурсы. Однако, необходимо правильно управлять памятью, чтобы избежать утечек или ошибок в работе с памятью. Поэтому важно освобождать память после использования и следить за правильностью выделения и освобождения памяти для каждого узла дека.